CMPSC 122 Lab 14 Report

Code

Code for bst.h:

// Programmer: Yizhou Wang

// Section: 2

// Lab: 14 - Binary Search Trees

// Date: April 8, 2014

// Description: A practice to implement binary search tree (bst) with class

#include <iostream>

using namespace std;

const int LOGICALSIZE = 100; //the maximum number of nodes a bst can have

struct TreeNodeType; //a new data type TreeNodeType

typedef TreeNodeType \* TreeNodePtr; //the type of what TreeNodePtr points to is

// TreeNodeType

struct TreeNodeType

{

int key; //value stored in this node

TreeNodePtr left; //pointer pointing to the left child of the node

TreeNodePtr right; //pointer pointing to the right child of the node;

};

class bst

{

public:

bst();

//POST: a default bst object is constructed with subroot = NULL

bst(int a[], int size);

//PRE: a[0..size-1] is initialized, all members of a[0..size] are unique

// 0 <= size <= LOGICALSIZE

//POST: a bst object is constructed from a[]

void insert\_helper(int value);

//PRE: value is initialized and distinct from other values stored in this bst

//POST: value is inserted into this bst at the right place

bool searchBST1\_helper(int value);

//PRE: value is initialized

//POST: FCTVAL = true if value is found in this bst

// FCTVAL = false if value cann't be found in this bst

void searchBST2\_helper(int value);

//PRE: value is initialized

//POST: the process of searching is printed to the console in a stepwise fashion,

// the result of the search is printed to the last line

void traverseBST\_inorder\_helper();

//POST: an inorder traversal of this bst is printed to the console

void traverseBST\_postorder\_helper();

//POST: a postorder traversal of this bst is printed to the console

private:

TreeNodePtr subroot; //a pointer to the subroot of this bst

void insert(TreeNodePtr & subroot,int value);

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys

// or is a NULL, value is initialized and distinct from other values stored in this bst

//POST: value is inserted into this bst at the right place

bool searchBST1(TreeNodePtr subroot, int value);

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys

// or is a NULL, value is initialized

//POST: FCTVAL = true if value is found in this bst

// FCTVAL = false if value cann't be found in this bst

void searchBST2(TreeNodePtr subroot, int value);

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys

// or is a NULL, value is initialized

//POST: the process of searching is printed to the console in a stepwise fashion

// the result of the search is printed to the last line

void traverseBST\_inorder(TreeNodePtr subroot);

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys

// or is a NULL

//POST: an inorder traversal of this bst is printed to the console

void traverseBST\_postorder(TreeNodePtr subroot);

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys

// or is a NULL

//POST: a postorder traversal of this bst is printed to the console

};

**Code for bst.cpp:**

// Programmer: Yizhou Wang

// Section: 2

// Lab: 14 - Binary Search Trees

// Date: April 8, 2014

// Description: A practice to implement binary search tree (bst) with class

#include "bst.h"

bst::bst()

//POST: a default bst object is constructed with subroot = NULL

{

//initialize the subroot

subroot = NULL;

}

bst::bst(int a[], int size)

//PRE: a[0..size-1] is initialized, all members of a[0..size] are unique

// 0 <= size <= LOGICALSIZE

//POST: a bst object is constructed from a[]

{

//initialize the subroot

subroot = NULL;

for (int i=0;i<size;i++) //insert all the elements of a[] into this bst

{

insert\_helper(a[i]);

}

}

void bst::insert\_helper(int value)

//PRE: value is initialized and distinct from other values stored in this bst

//POST: value is inserted into this bst at the right place

{

insert(subroot, value);

}

void bst::insert(TreeNodePtr & subroot, int value)

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys or is a

// NULL, value is initialized and distinct from other values stored in this bst

//POST: value is inserted into this bst at the right place

{

if (subroot == NULL) //if subroot is NULL, initialize the subroot node of this bst

// with key = value, left = NULL and right = NULL

{

subroot = new TreeNodeType;

subroot->key = value;

subroot->left = NULL;

subroot->right = NULL;

}

else if (value < subroot->key) //if value is smaller than key of the current node of this bst,

// go to the left branch

{

insert(subroot->left, value);

}

else //if value is greater than key of the current node of this bst,

// go to the right branch

{

insert(subroot->right, value);

}

}

bool bst::searchBST1\_helper(int value)

//PRE: value is initialized

//POST: FCTVAL = true if value is found in this bst

// FCTVAL = false if value cann't be found in this bst

{

return searchBST1(subroot, value);

}

bool bst::searchBST1(TreeNodePtr subroot, int value)

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys or is

// a NULL, value is initialized

//POST: FCTVAL = true if value is found in this bst

// FCTVAL = false if value cann't be found in this bst

{

if (subroot == NULL) //if this bst is NULL or value is not found

// in this bst

{

return false;

}

else if (subroot->key == value) //if value is found at the current node of

// of this bst

{

return true;

}

else if (value < subroot->key) //if value is smaller than the current node

// of this bst

{

return searchBST1(subroot->left, value);

}

else //if value is greater than the current node

// of this bst

{

return searchBST1(subroot->right, value);

}

}

void bst::searchBST2\_helper(int value)

//PRE: value is initialized

//POST: the process of searching is printed to the console in a stepwise fashion,

// the result of the search is printed to the last line

{

return searchBST2(subroot, value);

}

void bst::searchBST2(TreeNodePtr subroot, int value)

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys or is

// a NULL, value is initialized

//POST: the process of searching is printed to the console in a stepwise fashion

// the result of the search is printed to the last line

{

if (subroot == NULL) //if this bst is NULL or value is not found

// in this bst

{

cout << value <<" does not exist in the BST."

<<" Try a different value." << endl;

}

else if (subroot->key == value) //if value is found at the current node of

// of this bst

{

cout << "Checking " << subroot->key << " node."

<< "Does match " << value

<< ". So we find the key." << endl;

}

else if (value < subroot->key) //if value is smaller than the current node

// of this bst

{

cout << "Checking " << subroot->key << " node."

<< "Does not match " << value << ". "

<< value << " < " << subroot->key

<< ". So go left" << endl;

searchBST2(subroot->left, value);

}

else //if value is greater than the current node

// of this bst

{

cout << "Checking " << subroot->key << " node."

<< "Does not match " << value << ". "

<< value << " > " << subroot->key

<< ". So go right" << endl;

searchBST2(subroot->right, value);

}

}

void bst::traverseBST\_inorder\_helper()

//POST: an inorder traversal of this bst is printed to the console

{

cout << "Inorder Traversal of the binary search tree: " << endl;

traverseBST\_inorder(subroot);

cout << endl;

}

void bst::traverseBST\_inorder(TreeNodePtr subroot)

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys or is

// a NULL

//POST: an inorder traversal of this bst is printed to the console

{

if(subroot != NULL) //Not an empty tree node

{

traverseBST\_inorder(subroot -> left);

cout << subroot -> key << " ";

traverseBST\_inorder(subroot -> right);

}

}

void bst::traverseBST\_postorder\_helper()

//POST: a postorder traversal of this bst is printed to the console

{

cout << "Postorder Traversal of the binary search tree:" << endl;

traverseBST\_postorder(subroot);

cout << endl;

}

void bst::traverseBST\_postorder(TreeNodePtr subroot)

//PRE: subroot points to the subroot node of a subtree of this bst containing unique keys or is

// a NULL

//POST: a postorder traversal of this bst is printed to the console

{

if(subroot != NULL)

{

traverseBST\_postorder(subroot->left);

traverseBST\_postorder(subroot->right);

cout << subroot -> key << " ";

}

}

**Code for main.cpp:**

// Programmer: Yizhou Wang

// Section: 2

// Lab: 14 - Binary Search Trees

// Date: April 8, 2014

// Description: A practice to implement binary search tree (bst) with class

#include <iostream>

#include "bst.h"

using namespace std;

void treeSort(int data[], int size)

//PRE: b[0..size-1] is initialized and 0 <= size <= LOGICALSIZE

//POST: elements of b[] are printed inorderly to the console in one raw

{

bst test1(data, size);

test1.traverseBST\_inorder\_helper();

}

int main()

{

//DATA DICTIONARY

int a[8]; //data array for test1

//initilization of test1

a[0]= 5;

a[1]= 10;

a[2]= 3;

a[3]= 0;

a[4]= 20;

a[5]= -3;

a[6]= 4;

a[7]= 9;

bst test1(a,8);

cout << "Here below is the array data for test1: " << endl;

for (int i=0; i<8; i++)

{

cout << a[i] << " ";

}

cout << endl;

//test the default constructor

bst test2;

//test the insert method

test2.insert\_helper(5);

test2.insert\_helper(8);

test2.insert\_helper(-1);

//test the searchBST1\_helper and searchBST2-helper method

cout << "Is 9 in test1?" << endl;

cout << boolalpha << (test1.searchBST1\_helper(9));

cout << endl;

cout << "Here below are are the stepwise comparisons:" << endl;

test1.searchBST2\_helper(9);

cout << endl;

//test the traverseBST\_inorder\_helper method and traverseBST\_postorder\_helper

cout << "Here below is the inorder traversal of test2" << endl;

test2.traverseBST\_inorder\_helper();

cout << endl;

cout << "Here below is the postorder traversal of test2" << endl;

test2.traverseBST\_postorder\_helper();

cout <<endl;

//test for treeSort

cout << "Here below is the original array of test1:" << endl;

for (int i=0; i<8; i++)

{

cout << a[i] << " ";

}

cout << endl;

cout << "Here below is the array after treeSort:" << endl;

treeSort(a, 8);

return 0;

}

Sample Runs

Here below is the array data for test1:

5 10 3 0 20 -3 4 9

Is 9 in test1?

true

Here below are are the stepwise comparisons:

Checking 5 node.Does not match 9. 9 > 5. So go right

Checking 10 node.Does not match 9. 9 < 10. So go left

Checking 9 node.Does match 9. So we find the key.

Here below is the inorder traversal of test2

Inorder Traversal of the binary search tree:

-1 5 8

Here below is the postorder traversal of test2

Postorder Traversal of the binary search tree:

-1 8 5

Here below is the original array of test1:

5 10 3 0 20 -3 4 9

Here below is the array after treeSort:

Inorder Traversal of the binary search tree:

-3 0 3 4 5 9 10 20

Discussion

At the beginning, I struggled with the idea of implementing the tree via a class. The tips for the lab really helped as I could play with the hardcoded binary search tree. As I got used to the data structure and successfully implemented the search method outside, I was confident to combing class the the TreeNodeType. Nitish helped me find the private data, which is the pointer to the binary search tree. Then things get more interesting as I learned how helper methods call their corresponding private methods with less parameter. Lastly, the TreeSort is a great integration of the methods we developed in this lab.

I feel that C++ is a very powerful and versatile language. Things just get interesting (or sometimes puzzling) when pointers come to the stage.